Leap Forward Virtual Clock:
A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness

Subhash Suri George Varghese Girish Chandranmenon

Department of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

We describe an efficient fair queuing scheme, Leap Forward Virtual Clock, that provides end-to-end delay bounds similar to WFQ, along with throughput fairness. Our scheme can be implemented with a worst-case time $O(\log\log N)$ per packet (inclusive of sorting costs), which improves upon all previously known schemes that guarantee delay and throughput fairness similar to WFQ. Interestingly, both the classical virtual clock and the Self-Clocked Fair Queuing schemes can be thought of as special cases of our scheme, by setting the leap forward parameter appropriately.

1 Introduction

The study of service disciplines in integrated services networks has gained significance with the emergence of multimedia applications and electronic commerce. Unlike traditional data applications, real-time applications such as video-on-demand and teleconferencing must meet stringent quality-of-service (QoS) standards to be successful. In an integrated services network, applications with widely different communication rates (ranging from few bits to several megabits per sec), traffic patterns, and QoS requirements share communication links. This makes it difficult to provide good bounds on end-to-end delay and bandwidth.

With many delay-sensitive traffic sources contending for bandwidth, a service discipline must provide a fast and effective mechanism for servicing packets so that real-time traffic is not affected by non-time-critical traffic such as file transfer. Also, both kinds of traffic must receive a (preset) fair share of bandwidth. A lack of "firewalls" can result in a rogue traffic source (either a misbehaving user or malfunctioning device) unjustly penalizing well-behaved sources. In today's Internet, for instance, a rogue user can send data at an uncontrolled rate, and seize a large fraction of the available bandwidth at the expense of "good" users. Weighted Fair Queuing(WFQ) [5] is a benchmark for comparing properties of packet schedulers. WFQ, however, requires $O(N)$ time to schedule a packet, where N is the number of concurrent flows at a router. Other algorithms [14, 2] equivalent to WFQ require $O(\log N)$ time per packet. Our main result is a service discipline called *Leap Forward Virtual Clock (LFVC)* whose delay bound and throughput fairness is almost identical to WFQ, but whose computational overhead is only $O(\log\log N)$ per packet. Our $O(\log\log N)$ time bound is worst-case and includes all sorting overheads[1]. $O(\log\log N)$ is a small constant for all practical purposes—for instance, $\log\log N \leq 5$ for all $N \leq 4 * 10^9$; the underlying constants are also quite small.

Bennett and Zhang[1] introduced a more refined form of fairness called worst-case fairness (WFI). They showed that even WFQ could exhibit burstiness (i.e., high WFI), and described a scheme called WF^{2}Q that has optimal WFI. Our LFVC scheme has a WFI comparable to WF^{2}Q. Thus Leap Forward Virtual Clock appears to be the first service discipline that achieves near-ideal delay and throughput bounds, but has a computational overhead smaller than the $O(\log N)$ of prior schemes. We summarize the salient features of previous algorithms in Table 1, and compare them to our Leap Forward scheme. A detailed comparison can

[1] in contrast with some existing algorithms whose $O(1)$ time complexity bound accounts only for "tag computation" and not the additional $O(\log N)$ cost for sorting

0-8186-7780-5/97 $10.00 © 1997 IEEE

Scheme	Delay Bound	Fairness	Worst-case Fairness	Efficiency
GPS[10]	0	Fair	Excellent	Impractical
WFQ[5]	Small	Fair	Poor	$O(N)$
SCFQ[9]	Large	Fair	Poor	$O(\log N)$
Virtual Clock[17]	Small	Unfair	Poor	$O(\log N)$
Deficit Round Robin[12]	Large	Fair	Poor	$O(1)$
Frame Based FQ[14]	Small	Fair	Poor	$O(\log N)$
WF^2Q[1]	Small	Fair	Good	$O(\log N)$
Leap Forward VC	Small	Fair	Good	$O(\log \log N)$

Table 1: A comparison of several well-known scheduling algorithms. By a "small" delay, we mean a delay that is only a small additive constant larger than GPS delay, while "fair" throughput is fairness comparable to that of SCFQ or GPS. By poor worst-case fairness, we mean a worst-case index that grows with the number of flows.

be found in [15].

The paper is organized as follows. In Section 2, we formulate two conditions which are necessary and sufficient to guarantee WFQ-like delay and throughput bounds. In Section 3, we introduce our main algorithm, Leap Forward Virtual Clock (LFVC). We prove end-to-end delay bounds in Section 4, and throughput fairness in Section 5. In Section 6, we describe tag coarsening and how it leads to an $O(\log \log N)$ time implementation. In Section 7, we report on simulation experiments, and conclude in Section 8.

2 Virtual Clock Service Disciplines

Service disciplines based on a virtual clock work by assigning a tag (deadline) to each packet, representing the clock value by which the packet must be transmitted. Packets are serviced in non-decreasing order of tags. It is true but not obvious [7, 16] that such a scheme meets all tag deadlines. We have isolated two fundamental conditions that underlie delay and throughput guarantees. Our Leap Forward Virtual Clock scheme is an instance of a class of service disciplines that derive from these conditions. In particular, by setting parameters appropriately, both classical virtual clock [17] and Self-Clocked Fair Queuing [9] can be derived as special cases.

We begin with some useful notation. A *flow* is a logical connection between a source and a destination. Each packet in a flow carries the ID of the flow. Packets in a flow pass through a sequence of servers (or routers) along their path. We will later use the framework of *guaranteed rate clock* (GRC) algorithms introduced by Goyal et al. [8], to establish WFQ-like bounds on end-to-end delay. With this framework, it suffices to establish the delay bound at a single server. Hence, we concentrate on a server S, with output rate B bits per second.

Let $f_1, f_2, \ldots, f_N$ denote the set of flows that are serviced at S, where flow f_i has a guaranteed rate of r_i bits/sec, and $\sum_{i=1}^{N} r_i \leq B$. The sequence of packets in a particular flow f is denoted $p_f^1, p_f^2, \ldots, p_f^k$, and their sizes (in bits) are denoted $\ell_f^1, \ell_f^2, \ldots, \ell_f^k$. We also use the notation $\ell(p)$ to denote the length of p. We assume S is work-conserving: S does not idle when there is some packet to send. The virtual clock associated with the server is a counter that keeps track of the bits sent out by S. The output rate of S is B bits/sec, and thus servicing a packet p of length ℓ increments the clock by ℓ/B. Every packet to be serviced by S receives a *tag*, indicating the server clock value by which it must be serviced. The tag of the jth packet in flow f is denoted $T(p_f^j)$.

Let t_s denote the *current* server time (clock value) at any instant. The arrival time of a packet p_f^j, denoted $A(p_f^j)$, is defined as the server time when p_f^j reaches the head of its flow queue. Let t_f^{prev} denote the tag of the last packet sent by f; it is zero if no packet of f has been sent. We define the *tag* of p_f^j as follows:

$$T(p_f^j) = \max\{A(p_f^j),\ T(p_f^{j-1})\} + \frac{\ell_f^j}{r_f} = \max\{t_f^{prev},\ t_s\} + \frac{\ell_f^j}{r_f}. \quad (1)$$

Next, we define the *current tag* of a flow f, denoted t_f. We say a flow f is *active* if it currently has at least one packet in its queue. Otherwise, f is called *idle*. If f is active, let p_f denote the packet at the head of f's queue, and l_f the length of p_f in bits; this is the current packet for f. The current tag of f is defined as follows:

$$t_f = \begin{cases} T(p_f) & \text{if } f \text{ is active} \\ \max\{t_f^{prev}, t_s\} & \text{otherwise} \end{cases}$$

Finally, we define an important parameter Δ_f for each flow f:

$$\Delta_f = \frac{\ell_f^{max}}{r_f}, \qquad (2)$$

where ℓ_f^{max} is the size of the largest packet in flow f. Note that Δ_f is the time needed to send the largest packet by a flow at its guaranteed rate. Lastly, we use the notation τ to denote the time to transmit a largest packet at the server rate; thus, $\tau = \frac{M}{B}$, where M is the size of the largest packet across all flows.

2.1 Delay Bound for Virtual Clock

Classical virtual clock[17] assigns a tag to each arriving packet using Eq. (1), and services packets in non-decreasing tag order. If one can show that each packet is transmitted no later than its tag value, then the GRC framework of Goyal et al. [8] provides a good end-to-end delay bound. We distill a fundamental invariant, called *Backlog Inequality*, that is essential to ensuring that a packet is serviced by its tag. Our analysis is new and simple; more importantly, it leads directly to our new scheme.

Let t be an arbitrary server time such that $t \geq t_s$ and consider the time window (t_s, t). The Backlog Inequality relates the backlog (both packets waiting and potential future arrivals) that must be serviced within the time window (t_s, t) and the amount of bandwidth available: $B \times (t - t_s)$. A service discipline can meet all tag commitments only if the available bandwidth is no less than the backlog. Fig. 1 depicts the inequality.

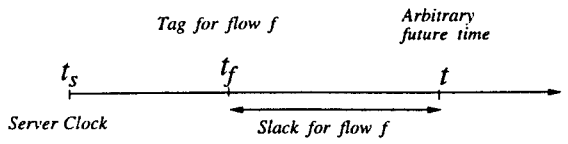

Figure 1: The Backlog Inequality.

More formally, let Φ_t denote the set of all flows whose current tags have value at most t:

$$\Phi_t = \{f \mid t_f \leq t\}.$$

Thus Φ_t is the set of flows whose most recent packet to receive service has tag less than t. These are the flows whose packets might need to be serviced before server clock reaches t.

[Backlog Inequality:] Given a server S with output rate B, the current server time t_s and a future server clock value t, we say that the Backlog Inequality holds if:

$$\sum_{f \in \Phi_t} \ell_f + \sum_{f \in \Phi_t} r_f(t - t_f) \leq (t - t_s) \times B \qquad (3)$$

On the left hand side, the first term is the current backlog, and the second term is the potential backlog or slack. The right hand side represents the available bandwidth in this time window. Later, we will show that *any service discipline that maintains the Backlog Inequality guarantees a WFQ-like delay bound.*

2.2 Delay and Throughput Conditions

The virtual clock discipline is known to suffer from throughput unfairness. The problem is caused by unbounded deviations between the server clock and a flow tag, which can happen when some flows send packets in bursts while others remain idle. (See [13] for examples.) Surprisingly, Bennett and Zhang [1] show that even WFQ exhibits a form of bursty behavior. Leap Forward Virtual Clock avoids both forms of unfairness. We show that a service discipline should satisfy the following delay and throughput conditions to deliver WFQ-like delay bounds and a good worst-case fairness index.

[Delay Condition:] A service discipline satisfies the delay condition if it satisfies the Backlog Inequality at all times.

[Throughput Condition:] Let k be a constant independent of N. A service discipline satisfies the throughput condition if each flow f satisfies the following condition at all times:

$$t_f \leq t_s + k \cdot \Delta_f.$$

These conditions are necessary for a virtual clock scheme to guarantee delay and throughput bounds similar to WFQ. Later, we establish the sufficiency of these conditions. These conditions motivate our Leap Forward algorithm, which modifies classical virtual clock by introducing two key ideas (quarantine and leap forward).

3 Leap Forward Virtual Clock

Our algorithm uses a strategy employed by parents for disciplining children: take them temporarily offline. For us, a flow in possible need of discipline is one that has recently received more than its allocated share of the bandwidth; the tag of such a flow might violate the Throughput Condition. We call such a flow *oversubscribed.* Any packets for an oversubscribed flow are placed in a holding area L

(see Figure 2). Other well-behaved flows are placed in a priority queue H. The server picks the packet with the lowest current tag in H for servicing next. We note that the lowest tag in H may very well be bigger than the lowest tag in L, yet the packet from H is serviced first. The data structures H and L can be thought of as *High* and *Low* priority queues.

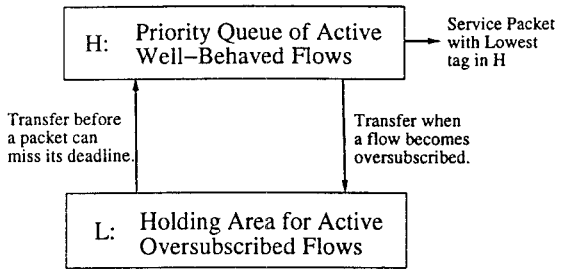

Figure 2: A conceptual picture of Leap Forward Scheme.

Just as children are forgiven after a "time off" period, we need to eventually transfer an active flow from L to H. How long can we wait before doing so? We must transfer f before we violate the delay condition. If f is a flow in L and p is the packet at the head of its queue, a safe test for keeping f in L is:

$$T(p) - t_s \geq \Delta_f.$$

We must transfer f to H before this condition is violated. We show that this transfer policy maintains the delay condition, and every packet is serviced no later than its tag.

Thus, our scheme transfers a flow to L when it is likely to cause a throughput problem, and restores it to H before the delay guarantee of its packet can be violated. There remains, however, one problem: what happens if all flows are oversubscribed, and therefore H is empty? To be work-conserving, we must transmit a packet; yet there is no eligible packet. This brings us to the second important feature of our algorithm. When H is empty, we *advance the server clock as far forward as possible without violating the delay invariant of any flows in L.*

We can prove that the Delay Condition for a flow f in L is satisfied if the following holds: $t_f - t_s \geq \Delta_f$. We call our scheme Leap Forward Virtual Clock because the server clock can leap forward when all active flows are oversubscribed. After the leap forward step, at least one active flow in L becomes eligible for transfer to H. Thus the server is work-conserving.

The leap forward step can be implemented by advancing the server clock to the smallest value of $(t_f - \Delta_f)$ among all active flows f that are in L. This suggests organizing the flows in L as a priority queue with key $t_f - \Delta_f$ (instead of t_f as in H). The use of these offset flow tags allows us to check the Leap Forward and Transfer conditions only for the flow(s) at the head of L.

3.1 Transfer Thresholds

The description above can be used to generate a family of schemes. A particular implementation involves fixing certain parameters. For instance, maintaining the Delay Condition requires there be "sufficient time" left for the deadlines of all flows in L. In our implementation, we ensure that any flow f in L satisfies the following *No Rush* condition:

$$t_f - \Delta_f \geq t_s.$$

Any flow f in violation of the "No Rush" condition must be transferred to H. Since L is a priority queue with $t_f - \Delta_f$ as keys, it suffices to check this condition for the flow g at the head of L. If g violates the No Rush condition, we transfer it to H, and repeat the process on the new flow at the head of L, until we reach a flow that meets the condition. At this point, we can be sure that all flows remaining in L satisfy the condition. While we may have to perform multiple transfers from L to H, each packet is extracted from L at most once. Thus, the amortized cost per packet is at most one Delete operation from L.

Finally, since server clock advances discretely, all tests use an additional slack parameter $\tau = \frac{M}{B}$, where M is the largest packet in the system. To make the Throughput Condition concrete, we also fix $k = 2$.

3.2 Allowing Rounded Tags

The Internal Revenue Service allows tax filers to round numbers to the nearest dollar. The intent is reduced computation at the expense of a possible small loss. We investigate a similar tradeoff between accuracy of tags and computational overhead. Let ρ be an arbitrary number. We will "round" up tag values to a whole multiple of ρ. Implementing the Leap Forward scheme with exact tags corresponds to setting $\rho = 0$.

Fix a value of ρ. The coarsened version of Leap Forward continues to compute fully accurate tags t_f and t_s, but rounds them up to the nearest multiple of ρ *before inserting them into the priority queues H or L.* Since rounding may introduce an error of ρ in the relative values of two tags, all our tests, such as Throughput and Delay Conditions, can be off by ρ. We prove that this uncertainty increases the de-

lay bound by at most ρ, and worsens the worst-case throughput fairness index by a small amount.

3.3 Leap Forward Code

Leap Forward Virtual Clock

ProcessHead (Q_f)
  if $(Q_f$ *is empty*$)$ **return**;
  $p \leftarrow head(Q_f)$;
  $t_f \leftarrow T(p) \leftarrow \max(t_s, t_f) + \frac{\ell(p)}{r_f}$;
  if $(t_f \leq t_s + \Delta_f + \tau + \rho)$ **then**
    $Insert(H, p, t_f)$);
  else $Insert(L, p, t_f - \Delta_f)$;
End.

Enqueue (* A new packet p_f arrives. *)

AddToTail (Q_f);
if $(Q_f$ was empty$)$ **then**
  ProcessHead (Q_f);

Dequeue (* The server is idle and there is a packet in the system. *)

$k_{min} \leftarrow MinKey(L)$;
Let f be the flow corresponding to the key k_{min}.
if $(H$ *is empty*$)$ **then**
  $t_s = \max(t_s, k_{min} - \rho)$ (* Leap forward *)
while $(k_{min} < t_s + \tau + \rho)$
  $p_f \leftarrow ExtractMin(L)$;
  $Insert(H, p_f, T(p_f))$; (* Transfer *)
  $k_{min} \leftarrow MinKey(L)$;
end
$p_f \leftarrow ExtractMin(H)$;
$t_s \leftarrow t_s + \frac{\ell(p_f)}{B}$. (* Start Service *)
RemovefromTail (Q_f)
ProcessHead (Q_f);
Transmit p_f; (* Real time elapses *)

Figure 3: Leap Forward Virtual Clock: Algorithm

The Leap Forward algorithm asynchronously performs two operations repeatedly: Enqueue and Dequeue. The former handles the task of inserting a newly arrived packet in appropriate data structures, while the latter deals with prioritizing packets for service. The server always picks for transmission the packet p with the current smallest tag in H. The server clock is advanced by $\ell(p)/B$, and p is deleted from the flow queue. Only then does the server begin transmitting the packet p. Real time elapses only during packet transmission. All other events are assumed to be instantaneous. The server clock is incremented before the packet is actually transmitted!

The code for Leap Forward algorithm is described in Figure 3. The algorithm maintains two priority queues, H and L, which allow *Insertion* and *Extract-Min* operations. Each flow f has a FIFO queue Q_f which contains flow f packets that are waiting for service. t_f maintains the flow tag of f, while t_s is the server clock. Variables t_s and t_f, for all f, are initialized to 0, and reset to zero whenever the server becomes idle.

The procedure $Insert(\mathcal{A}, p, x)$ inserts the *rounded* value of x (assuming some fixed coarsening parameter ρ) into the priority queue $\mathcal{A}$. The value x is the tag of p if $\mathcal{A} = H$ and the offset tag if $\mathcal{A} = L$. In addition, the procedure *ProcessHead* is used to handle a packet p when it reaches the head of its queue Q_f.

We show that:

Theorem 3.1 *LFVC is a work-conserving service discipline.*

Due to lack of space, we omit the proof. Please refer to [15] for details.

4 Delay Bounds for LFVC

We establish the delay bound at a single server, and extend it to multiple hops.

4.1 Delay at a Single Server

We show that LFVC maintains the Backlog Inequality. Due to our use of rounded tags, we only require that this Inequality hold for times t that are whole multiples of the parameter ρ. The following lemma collects some useful invariants on flow tags.

Lemma 4.1 (Tag invariants) *Consider a flow f, and its current tag t_f. LFVC maintains the following invariants with respect to the current server time t_s:*

T1: *If t_f is in H, then $t_f \leq t_s + \Delta_f + \tau + \rho$.*

T2: *If t_f is in L then $t_f \geq t_s + \Delta_f$.*

T3: *Let ℓ_f be the length of packet at the head of f's queue (assume $\ell_f = 0$ if f is idle). Then, $t_f \leq t_s + \tau + \rho + \Delta_f + \ell_f/r_f$.*

Our next lemma proves a weaker version of the backlog inequality, which we use in establishing our final lemma. Let us recall the definition of the set Φ_t from earlier discussion: $\Phi_t = \{f \mid t_f \leq t\}$ is the set of flows whose current tags are less than t, for a server time $t \geq t_s$. We call a flow *backlogged* if it currently has at least one packet in its queue.

Lemma 4.2 (Sufficient Bandwidth) *If, for all $t \geq t_s$, every backlogged flow $f \in \Phi_t$ satisfies $\ell_f \leq (t_f - t_s) r_f$, then the backlog inequality holds at t_s.*

Let us use the notation $R(n)$ to denote the value of n *rounded up* to the nearest multiple of ρ. Abusing the notation slightly, we also use $R(p)$ to denote $R(T(p))$. We are now ready to state and prove the following key lemma.

Lemma 4.3 (Backlog Lemma) *In LFVC, the Backlog Inequality holds for all rounded values of t. That is, if t is a whole multiple of ρ, then the following holds:*

$$\sum_{f \in \Phi_t} \ell_f + \sum_{f \in \Phi_t} (t - t_f)\, r_f \;\leq\; (t - t_s) \times B.$$

PROOF. Our proof is by induction on the number of *events*, where an event is either an Enqueue, or any of the following four steps of Dequeue: Leap Forward, Transfer, Start Service, and End Service. Due to lack of space, we only discuss the proof for the Enqueue; the remaining cases are similar and a complete proof can be found in the technical report [15].

[*Enqueue.*] Suppose a packet p_f arrives at the head of queue for the flow f. Let t_f be the current tag of f before p_f arrives. Then, the tag of p_f is given by $T(p_f) = t_f + \frac{\ell_f}{r_f}$. We consider two cases, depending upon whether or not $T(p_f) < t$. If $T(p_f) \geq t$, then $f \notin \Phi_t$, and the lemma clearly holds. If $T(p_f) < t$, then the first term on the left hand side of the Backlog Inequality increases by ℓ_f, but the second term *decreases* by $(t - t_f)\, r_f - (t - T(p_f))\, r_f = (T(p_f) - t_f)\, r_f = \ell_f$, and the right hand side remains unchanged, Hence the inequality is preserved.

□

The preceding lemma implies that at any time t the server has sufficient bandwidth to service packets scheduled before t. This allows us to bound the service time of a packet:

Lemma 4.4 (Service Time) *A packet p is completely transmitted by the time the server clock reaches $R(p) \leq T(p) + \rho$. Thus, no packet in H or L has a tag smaller than $t_s - \rho$.*

The preceding lemma bounds a packet's service time in terms of the virtual server clock. But we need a statement in terms of real time. Our next lemma establishes a connection between the server clock and real time. Let I be a variable that maintains the value of the real time when the server was last idle. If the server is currently busy, then I equals the real time when the last busy period began. If the server is idle, then I equals the current value of real time. Similarly, let L be a variable that maintains the amount of leap-forward done in the current busy period up to the current instant of time. At the start of a busy period, we set $L = 0$. The following lemma shows that real time equals server time plus $I - L$, except during packet transmission, when it can be behind by τ.

Lemma 4.5 (Server Clock Deviation) *Let t be real time and let I and L be defined as above. Then, the following holds:*

1. *During a packet transmission, $t \leq (t_s + I - L) \leq t + \tau$.*
2. *At all other times, $t = (t_s + I - L)$.*

The preceding two lemmas imply that a packet p is serviced in the Leap Forward system no later than the real time value $R(p) + I - L$. We use this fact to prove end-to-end delay bounds.

We then show that our scheme is a Guaranteed Rate scheduling algorithm, as defined by Goyal et al. [8]. This provides an end-to-end delay bound from our single-server delay bound; we omit details, and state the result.

Theorem 4.6 (End-to-end Delay Bound) *Suppose that a flow f conforms to a leaky bucket process with parameters (σ_f, r_f), and the scheduling algorithm at each of the K servers on its path is LFVC. Then, the end-to-end delay of a packet p_f^n, denoted by d_f^n, is given by the following:*

$$d_f^n \leq \frac{\sigma_f}{r_f} + (K-1) \max_{j=1}^{n} \frac{\ell_f^j}{r_f} + \sum_{i=1}^{K} \alpha^i.$$

Leap Forward has $\beta = \rho + \tau$. If we eliminate tag coarsening (set $\rho = 0$), our result is identical to the standard delay bound of classical virtual clock and WFQ. Tag coarsening adds an additional delay of at most ρ per server on the path. The parameter ρ can be used to derive tradeoffs between additional delay and savings in computational overhead.

Goyal et al. [8] also show how to obtain probabilistic bounds on the end-to-end delay when the burstiness of a flow is bounded with a stochastic process, such as the exponentially bounded burstiness. The same results also apply to our scheduling algorithms. In next section, we address the throughput fairness of our algorithm.

5 Throughput Fairness of Leap Forward

Informally, a scheme is fair if each backlogged flow receives its fair share of the available server bandwidth. There are two commonly used fairness measures: the first, proposed by Golestani [9], is a gross measure, while the second, introduced recently by Bennett and Zhang [1], is a more refined measure. Bennett-Zhang call their measure *worst-case fairness index* (WFI), and show that while WFQ is fair by Golestani's measure, it falls short of the ideal GPS by the worst-case fairness index. WFI more closely measures short-term bursty behavior. We will show that our Leap Forward Scheme satisfies both Golestani's as well as Bennett-Zhang's definition of fairness.

5.1 Fairness Measure of Golestani

A flow is said to be *backlogged* during interval (t_1, t_2) if the queue for flow f is never empty during (t_1, t_2). Let $sent_f(t_1, t_2)$ denote the total number of bits of f transmitted during (t_1, t_2) by the server. Throughput fairness of an algorithm is measured by the maximum (absolute) difference between rate-normalized values of $sent_f(t_1, t_2)$ and $sent_f(t_1, t_2)$ over all pairs of backlogged flows and over all intervals (t_1, t_2). In other words, consider an execution of the Leap Forward algorithm, and define $F(t_1, t_2)$ as follows:

$$F(t_1, t_2) = \max_{f,g} \left| \frac{sent_f(t_1, t_2)}{r_f} - \frac{sent_g(t_1, t_2)}{r_g} \right|,$$

where max is over all pairs of flows that are backlogged during (t_1, t_2). Then, throughput fairness is measured by the worst-case maximum value of $F(t_1, t_2)$ over all intervals and all executions of the algorithm:

$$F = \max_{(t_1, t_2)} F(t_1, t_2).$$

We call a service discipline *fair* if F is a small constant. In particular, F should be a constant, independent of the length of the time interval [9]. It is known that in the case of the classical virtual clock, $F \to \infty$. For LFVC, we prove:

Theorem 5.1 (Throughput Fairness) *The fairness measure of LFVC Clock satisfies:* $F \leq \max_{f,g} 3(\Delta_f + \Delta_g) + 2\tau + 4\rho$, *where* $\tau = M/B$ *and* ρ *is a tag coarsening parameter.*

5.2 Worst Case Fairness Index

The following measure of throughput fairness, due to Bennett and Zhang [2] provides a more refined measure of short term throughput unfairness than the Golestani measure. We start with some notation.

The delay $D(p)$ of a packet p is the real time that elapses between the arrival time of p and the time p is completely transmitted. The arrival time of a packet p is denoted by $A(p)$. Assume p belongs to flow f. Let $Q(p)$ denote the size (in bits) of the queue in front of p (including p) at the time of p's arrival.

Definition 5.2 *A service discipline is said to be worst-case for flow f if, for any packet p in f, the following holds:* $D(p) < \frac{Q(p)}{r_f} + C_f$, *where C_f is a constant independent of other flows sharing this server. The normalized worst-case fair index (WFI) is defined as* $C = \max_f \frac{r_f C_f}{B}$.

Bennett and Zhang [1] have shown that Worst-case Weighted Fair Queuing has an optimal WFI equal to $\tau = M/B$. We show that the WFI of our scheme is nearly the same. Our proof depends critically on two facts: one, the current tag of a backlogged flow differs from the server clock by no more than $2\Delta_f + \tau + \rho$ (Lemma 4.1), and, two, during a backlogged period (t_1, t_2), the difference of the initial and final tags of f is (roughly) the number of bits of flow f transmitted during (t_1, t_2) divided by its rate r_f.

Theorem 5.3 *The Worst-Case Fairness Index of Leap Forward Virtual Clock is bounded by* $\tau + \max_f 2(\tau + \rho) \frac{r_f}{B}$.

In practice, r_f/B should be quite small, and if we set the coarsening parameter ρ to roughly τ, the WFI of Leap Forward approaches τ, which is the optimal value of WFI [1].

6 Data Structures

The only nontrivial data structure needed for implementing LFVC is a priority queue. Using standard data structures for maintaining priority queues, we can implement LFVC in $O(\log N)$ time per packet. We can improve the processing cost to $O(\log \log N)$ per packet using two key ideas: *tag coarsening* and a *finite-universe priority queue.*

The basic idea of tag coarsening is that maintaining *exact* order among virtual clock tags is overkill if one is willing to tolerate a minor increase in latency. For instance, suppose the server rate is B and the largest packet size in any flow is M. Then, rounding up all the tags to multiples of M/B dramatically reduces the underlying key space of the priority queue, while increasing the delay by at most M/B. We further reduce the key space to a set of $O(N)$ integers, in the range [1, cN] for a fixed constant c, by using a tag-separation property of our algorithm and *modular arithmetic* to recycle tags. With these

ideas in place, we use "approximate sorting" and a finite-universe priority queue of van Emde Boas [6] to achieve $O(\log\log N)$ processing time per packet. Details can be found in [15].

The following theorem summarizes the main result of our paper.

Theorem 6.1 *LFVC provides a guaranteed delay bound comparable to WFQ, throughput fairness comparable to worst-case WFQ, and can be implemented with the worst-case overhead of $O(\log\log N)$ per packet, where N is the number of active flows.*

6.1 Preliminary Implementations

We have implemented the $O(\lg\lg N)$ priority queue. The constants are roughly 250 Sparc instructions per operation. We have found that a second scheme based on tries, outperforms the (unoptimized) Van Emde Boas structure for small values of N. The trie scheme views a K bit tag b bits at a time, and goes through a tree of arrays b bits at a time. If b is small we can avoid searching through empty array elements by keeping a bitmap and doing a table lookup to determine the lowest non-null array position. A trie-based implementation for values in the range $[0..2^{21}-1]$ requires worst-case 85 Sparc instructions for Insert and 183 instructions for ExtractMin. Note that for both the trie scheme and the Van Emde Boas scheme, reducing the size of the tag by coarsening is crucial for decreasing processing costs. We hope that further optimizations will reduce the cost to around 40 instructions for values of $N < 10000$.

7 Simulations

We used a stripped down version of the network simulator, freely available from Lawrence Berkeley Laboratories,[2] to carry out simulations. We implemented several fair-queuing algorithms and compared their performance to Leap Forward Virtual Clock scheme. The simulator allows us to examine packets that go through a link, and calculate delays of individual packets and overall bandwidth seen by each flow.

Due to space constraints, we show only one of the simulation results in Table 7 where we compare the delay experienced by different flows under four different fair queuing schemes. Average delay experienced by flows under LFVC are similar to those under VC and much smaller than those under DRR and SCFQ. Other results [15] illustrate the merits of LFVC over other fair queuing schemes.

[2] `http://www-nrg.ee.lbl.gov/ns/`

[3] The rate allocations in this experiment are borrowed from [13]; but the flows here send data at a constant bit rate as opposed to an on-off model used in their experiments.

Based on our experiments, we observe: **1)** LFVC rations the output link fairly among contending flows. The leap forward step prevents credit accumulation. **2)** Flows sending packets at or under their reserved rate experience very small delay in LFVC but the delay suffered by ill-behaved flows (on average) appears to be worse in Leap Forward. **3)** LFVC does not cause bursty behavior as can be exhibited by both SCFQ and WFQ.

8 Concluding Remarks

We believe there are several important contributions of this paper. The first contribution is the algorithm itself. Leap Forward Virtual Clock goes well beyond Virtual Clock by adding two non-trivial modifications: a quarantine mechanism and a leap forward mechanism. The resulting scheme provides throughput fairness, even in a worst-case sense, without compromising delay bounds. The algorithm is elegant and simple to implement, and should have practical appeal.

Second, our analytical techniques (i.e., separation of delay and throughput conditions) are new and simple. They may be useful for other schemes as well (WFQ, SCFQ etc.)

Third, the idea of trading off delay bounds for reduced computation, leads to an exponential speedup for our algorithm. The same ideas may well be applicable to other fair queuing schemes but requires careful proofs. We found a number of subtle bugs before we were able to make coarsening work for LFVC without compromising throughput fairness.

The *scalability* offered by $O(\log\log N)$ schemes holds promise as networks become larger and operate at increasingly higher speeds. Our preliminary implementation shows that the underlying constant is reasonably small (insert, delete, and successor/findmin operations cost between 200 and 300 Sparc instructions). We believe that with further optimization we can rival more conventional schemes (such as tries), for even small values of N.

Future work includes using LFVC in Hierarchical Fair Queuing systems and extending the scheme to provide jitter bounds. We also plan to implement LFVC in a real router testbed.

References

[1] J. Bennett and H. Zhang Worst-case Fair Weighted Fair Queuing. *Proc. INFOCOM*, 1995.

[2] J. Bennett and H. Zhang Hierarchical Packet Fair Queuing Algorithms. in *Proc. SIGCOMM '96.*

[3] R. Brown Calendar Queues: a fast O(1) Priority Queue Implementation for the Simulation Event Set Prob-

Flow	Reserved rate[3]	arrival rate	Delay (seconds)			
			DRR	SCFQ	VC	LFVC
0	0.5	0.498	0.00143	0.00112	0.000213	0.000213
1	0.0625	0.1	0.368	0.365	0.382	0.382
2	0.0625	0.62	0.00242	0.00526	0.00186	0.00203
3	0.0625	0.61	0.00243	0.00535	0.002	0.00211
4	0.078125	0.076	0.00307	0.00404	0.000881	0.000861
5	0.078125	0.076	0.0035	0.00354	0.0025	0.00228
6	0.078125	0.076	0.00371	0.00335	0.00217	0.00229
7	0.078125	0.076	0.00392	0.00356	0.00114	0.00125

Table 2: Average delay experienced by flows under Deficit Round Robin (DRR), Virtual Clock, Self-Clocked, and Leap Forward. In this experiment we use constant bit rate flows with a packet size of 53 bytes. All flows except flow 1 are operating under their reserved rate. All these queuing algorithms provide fair sharing of the bandwidth. Since the columns representing throughput used by each flow under different schemes are identical to their allocated rate, they are not enumerated here. On the other hand, the delay experienced by flow 0 for example, varies widely across the different queuing disciplines. DRR and SCFQ do not provide good delay bounds, as can be seen from the table. Average delay experienced by flows under LFVC are similar to those under VC and much smaller than those under DRR and SCFQ.

lem. *Communications of the ACM, Vol 31,10, October 1988.*

[4] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. *MIT Press, 1990.*

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Proc. Sigcomm '89*, 19(4):1–12, September 1989.

[6] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[7] N. Figuera and J. Pasquale. Leave-in-time: A new service discipline for real-time communication in a packet-switching data network. *Proc. Sigcomm '95*, September 1995.

[8] P. Goyal, S. S. Lam, and H. M. Vin. Determining End-to-End Delay Bounds in Heterogeneous Networks. In *Proceedings of Workshop on Network and OS Support for Audio-Video*, pages 287–298, April 1995.

[9] S. J. Golestani. A Self-Clocked Fair Queuing Scheme for High Speed Applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, April 1994.

[10] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control: The Single Node Case. In *Proc. of IEEE INFOCOM*, pp. 915–924, 1992.

[11] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. In *Proc. of IEEE INFOCOM*, pp. 521–530, 1993.

[12] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proc. of SIGCOMM*, pp. 231–242, 1995.

[13] D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *Technical Report UCSC-CRL-95-38, Dept. of Computer Engineering and Information Sciences, UC Santa Cruz*, July 1995.

[14] D. Stiliadis and A. Varma. Frame-based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks. *Proc. of ACM SIGMETRICS '96, May 1996.*

[15] S. Suri, G. Varghese and G. Chandranmenon. Leap Forward Virtual Clock: A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness. *Technical Report, Dept. of Computer Science, Washington University, Saint Louis*, 1996. Use the WWW link: `www.cs.wustl.edu/~suri/leapforward.ps` to access the report.

[16] G. G. Xie and S. S. Lam. Delay Guarantee of a Virtual Clock Server. *IEEE/ACM Transactions on Networking, December 1995*

[17] Lixia Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet-Switched Networks. *ACM Transactions on Computer Systems*, 9(2), May 1991.